Walking in a Triangulation

Olivier Devillers, Sylvain Pion, Monique Teillaud

Walking in a Triangulation*

Olivier Devillers† Sylvain Pion† Monique Teillaud†

Published in Internat. J. Found. Comput. Sci., 13:181–199, 2002

Abstract

Given a triangulation in the plane or a tetrahedralization in 3-space, we investigate the efficiency of locating a point by walking in the structure with different strategies. **keywords:** computational geometry, geometric computing, randomized algorithms, Delaunay triangulation.

1 Introduction

Given a triangulation $\mathcal{T}$ of n vertices in the plane and a point p, finding the triangle of $\mathcal{T}$ containing p is a fundamental problem in computational geometry. Several sophisticated structures exist to answer such location queries in optimal $O(\log n)$ time [1, 2] but they are often too complicated and some practitioners may prefer to implement simpler techniques, such as traversing the triangulation using adjacency relations between triangles. In the early years of computational geometry, authors already used such walking techniques and mentioned robustness problems near degeneracies, due to rounded computations [3, 4, 5]. This idea can be used directly to locate a point in a triangulation from a known starting point. It is also possible to choose a good starting point in some clever way [6, 7, 8].

There exist different strategies to find the triangle containing the query point p from the triangle containing a source point q. The *straight walk*, the simplest strategy, consists in visiting all triangles along the line segment qp (see e.g. [9]). A second strategy, *the orthogonal walk*, visits the triangles along an isothetic path moving from q to p by changing one coordinate at a time. Finally, we call *visibility walk* the following strategy, popular for the Delaunay triangulation: from a triangle t not containing p, we move to the neighbor of t through an edge e if the line supporting e separates t from p; there may be one or two such edges for a triangle t, if there are two we may move to any of these two neighbors. This walk is used for the Delaunay triangulation because in that case it can be proved that it actually reaches the right triangle [10, 11, 12]. In the case of an arbitrary triangulation, the walk may loop. We consider a variant of the visibility walk: the *stochastic walk* in which we decide that if we can choose between two neighbors of t, then the choice is done at random.

*This work was partially supported by the ESPRIT IV LTR Project No. 28155 (GALIA).

†http://www-sop.inria.fr/geometrica/ INRIA, BP93, 06902 Sophia Antipolis, France.

All these walking strategies generalize to higher dimensions.

The purpose of this paper is to study the performances of the different strategies from both theoretical and practical points of view, in R^2 and R^3. Hardly anything is known on this topic. The only theoretical result states that the number triangles visited by the straight walk in a Delaunay triangulation of n random points in the plane, to reach a point p from a point q is $O(|qp|\sqrt{n})$, where $|qp|$ denotes the distance from q to p [13, 14].

We are interested in counting not only the number of simplices visited by a walk, but also the cost of visiting one simplex. We consider the robustness issues raised by the implementation of the different strategies.

Section 2 defines the framework of this study. Then we give a detailed description of the different strategies (Sections 3, 4 and 5) in dimensions 2 and 3 together with complexity results. We prove in Section 5.2 that the stochastic walk actually has a zero probability of looping forever, in any dimension. In Section 6 we present some experimental results on the implementation of the different strategies.

2 Framework

Let S be a set of n points in $R^d, d = 2, 3$. We will consider triangulations (simplicial complexes) whose domain covers the whole convex hull of S. All the simplices of a triangulation are positively oriented.

Given such a triangulation $\mathcal{T}$ of S, we study different strategies to reach a query point p starting from a given starting vertex q of $\mathcal{T}$, walking in $\mathcal{T}$ by using adjacency relations between the simplices of $\mathcal{T}$.

It is not straightforward to decide which strategy is the best one. The paths followed by the different strategies have different lengths in terms of number of simplices. The number of evaluations of *predicates* (simple geometric questions) when visiting a given simplex also depends on the strategy, as well as the nature itself of the predicates involved.

There are theoretical results on the number of triangles visited by the straight walk in the plane, but nothing is known about the visibility walk.

The basic predicate in the straight walk (Section 3) and the visibility walk (Section 5) is the *orientation* predicate, which is defined over $d + 1$ points by the sign of a d dimensional determinant, expressed below for 2 and 3 dimensions respectively:

$$\textit{orientation}(\alpha, \beta, \gamma) = \textit{sign}\left(\begin{vmatrix} \beta_x - \alpha_x & \gamma_x - \alpha_x \\ \beta_y - \alpha_y & \gamma_y - \alpha_y \end{vmatrix} \right)$$

$$\textit{orientation}(\alpha, \beta, \gamma, \delta) = \textit{sign}\left(\begin{vmatrix} \beta_x - \alpha_x & \gamma_x - \alpha_x & \delta_x - \alpha_x \\ \beta_y - \alpha_y & \gamma_y - \alpha_y & \delta_y - \alpha_y \\ \beta_z - \alpha_z & \gamma_z - \alpha_z & \delta_z - \alpha_z \end{vmatrix} \right)$$

When two points have all but one coordinate equal, the expression of *orientation* simplifies to a determinant of dimension $d-1$. We take advantage of this in the orthogonal walk (Section 4), which uses mostly *comparisons of coordinates* in dimension 2 and more generally *lower dimensional orientation* predicates, which are faster and of

course more robust than the full dimensional *orientation* tests, since they involve lower degree computations. The orthogonal walk only uses the d dimensional *orientation* predicate a constant number of times.

The algorithms also use basic operations such as:

- `neighbor(t through pq)` returns the triangle sharing edge `pq` with the triangle `t`.
- `l=vertex of t, l≠q, l≠r;` chooses `l` as the third vertex of a triangle whose two vertices are already known.

(the same notation will be used in the pseudo-code given in Appendix).

These two operations are similar in 3 dimensions for the neighbor of a tetrahedron or the fourth vertex of a tetrahedron. They need a constant number of pointers access or comparisons; the exact number depends on the internal representation of the triangulation, which may be any variant of the DCEL or may be based on simplices or vertices as in CGAL [15].

3 Straight walk

3.1 2 dimensions

This method consists in traversing all the triangles of the triangulation $\mathcal{T}$ that are intersected by the line segment originating from a given vertex q of $\mathcal{T}$ and ending at the query point p. This is performed using the adjacency relations between the triangles.

More precisely, the algorithm first performs an initialization step: from one triangle incident to q we turn around q until a triangle intersected by the ray qp is found. During this initialization step, one orientation test is needed for each visited triangle and the number of visited triangles is at most the degree of q, thus at most n triangles.

Once the initialization step is completed, the straight walk really starts. At a given step of the walk, we traverse some triangle t, and the ray qp goes out of t through edge e. By testing on which side of e lies p, we decide if t contains p or if the walk must go on. In the later case, the walk goes to the neighbor of t through e and the new vertex of that triangle is located with respect to the line qp to decide by which edge of that triangle the ray qp goes out (see Figure 1-left). Therefore, the number of orientation tests performed for each visited triangle is exactly 2. The straight walk cannot visit the same triangle twice, thus the worst case length of a straight walk is at most the number of triangles of the triangulation which is less than $2n$.

Of course, visiting a linear number of triangles seems big, but the general idea of the walking strategy is that in practice you visit less triangles in a reasonable triangulation, although the $2n + O(1)$ bound is tight, as shown by Figure 1-right. In the special case of a Delaunay triangulation of evenly distributed points, the number of visited triangles during the walk is $O(|pq|\sqrt{n})$ [13].

We give a pseudo-code for a detailed description of the walk (see Appendix).

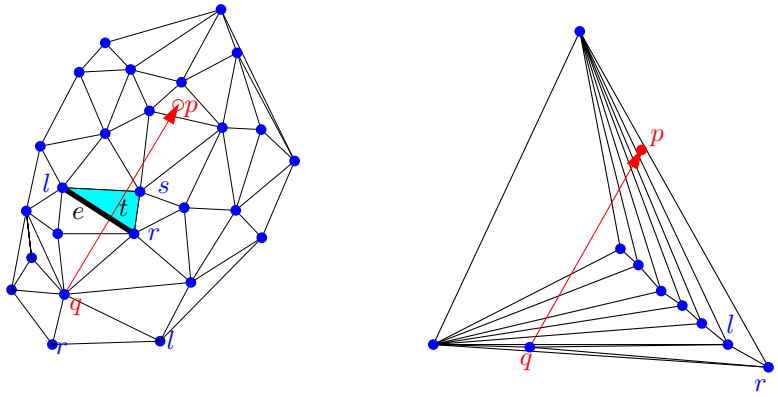

Figure 1: The straight walk.

3.2 3 dimensions

The principle of the walk is similar in higher dimension although a little bit more intricate.

Given vertex q of $\mathcal{T}$ and a query point p, the initialization step consists in finding the tetrahedron incident to q intersected by the ray qp, starting from another tetrahedron incident to q. This problem is in fact the 2 dimensional problem of locating the ray qp in the set of rays having q as origin and triangulated by the tetrahedra incident to q in $\mathcal{T}$. This initialization step is thus solved by the 2D Straight Walk algorithm. Notice that the orientation test for three rays emanating from q is the usual orientation test in three dimensions. The results of the previous paragraph on the number of visited triangles or the number of predicates per triangle apply here.

After this initialization, the main part of the walk begins. At a given step, we know that the ray goes out of some tetrahedron t by a facet e, then we must decide if the walk terminates in t by looking on which side of e lies p (see Figure 2). If the walk continues in the neighbor of t through e, then the ray qp goes out of that neighbor by a facet which is determined by two orientation tests involving q, p, the new vertex and a vertex of e.

Thus the number of orientation tests per visited tetrahedron is exactly 3. As in two dimensions, the number of visited tetrahedra is clearly bounded by the number of tetrahedra of $\mathcal{T}$ since a tetrahedron cannot be visited twice. This number is quadratic in the worst case and a quadratic bound may be reached as shown by the example of Figure 3.

3.3 Degenerate cases

The above algorithms do not handle degenerate cases. When the ray qp goes exactly through a vertex of the triangulation, or through an edge in 3D, the next cell traversed by the ray is not a neighbor of the previous one. In such a case, the algorithm must perform a kind of initialization step to be able to continue the walk.

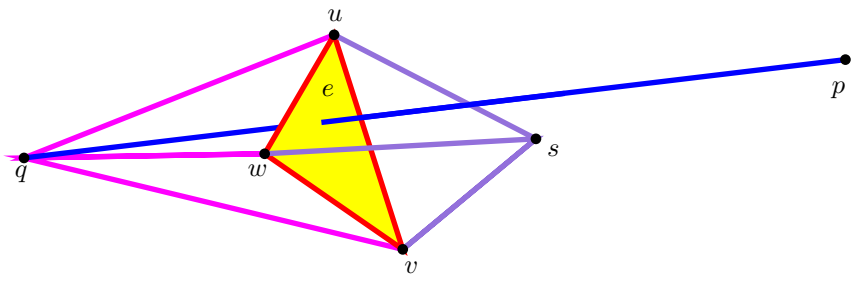

Figure 2: Straight walk in 3 dimensions (main loop).

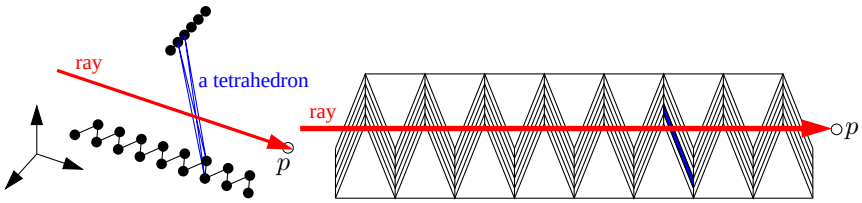

Figure 3: A quadratic example for the straight walk in three dimensions.

Actually coding a robust version of the straight walk which handles degenerate cases yields to an intricate code.

4 Orthogonal walk

The cost of evaluating an orientation predicate increases with the dimension, thus an idea to improve the efficiency of the algorithm consists in decomposing the walk in pieces parallel to the coordinate axis and to get an *orthogonal* walk (see Figure 4 left).

If the ray pq is parallel to a coordinate axes, then the orientation tests of the straight walk involving both p and q become simpler as noticed in Section 2. This is the case of the orientation tests involved in the initialization phase and of the tests to decide by which edge of the triangle (*resp.* facet of the tetrahedron) the ray goes out. It remains one test per triangle (*resp.* tetrahedron) to decide if the walk ends in that tetrahedron; this test cannot be simplified in general, but a cheaper sufficient condition for the ray to continue can be evaluated first: if p is further than the triangle bounding box in the axis direction, then the walk continues and only otherwise the orientation test is performed.

In the worst case, the orthogonal walk can visit the same simplex at most d times, thus the worst case length of an orthogonal walk is trivially linear in the number of simplices of the triangulation. A bound of $4n + O(1)$ can be reached in 2 dimensions (Figure 4 right). The orthogonal walk can be quadratic in 3 dimensions as shown by the example of Figure 3.

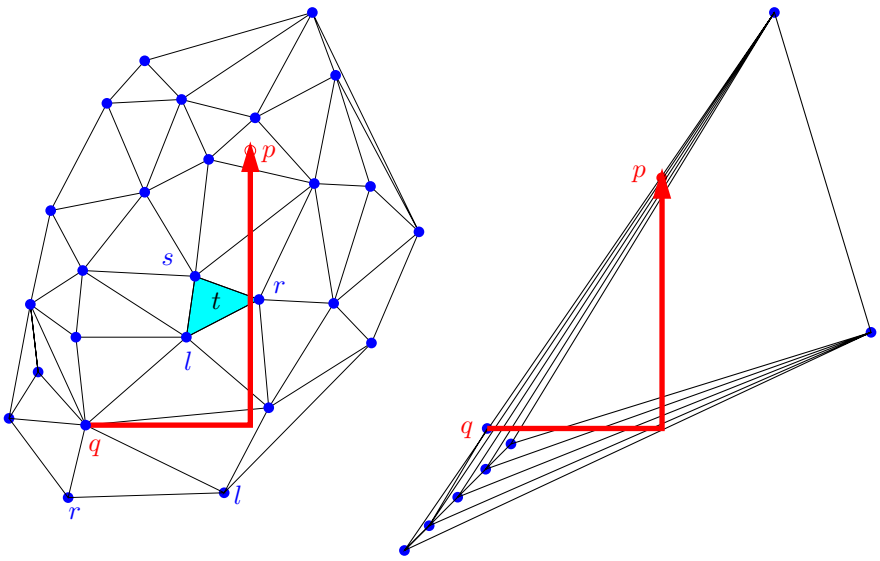

Figure 4: The orthogonal walk.

For the special case of the Delaunay triangulation of random points in the plane, the number of visited triangles during the walk is $O((|p\alpha| + |\alpha q|)\sqrt{n})$ [13]. In the orthogonal walk, the dimension of the orientation tests decreases, compared to the straight walk, but the number of visited triangles increases. The average ratio between the length of the straight and the orthogonal walks is the average, on the unit sphere in d dimensions, of the sum of the absolute values of the coordinates, which is d times the average of the absolute value of one coordinate, which we show below to be $4/\pi \approx 1.27$ in 2 dimensions, and $3/2$ in 3 dimensions.

In 2 dimensions :

$$\frac{2}{2\pi}\int_0^{2\pi} |\cos\theta|\, d\theta = \frac{8}{2\pi}\int_0^{\frac{\pi}{2}} \cos\theta\, d\theta = \frac{4}{\pi}\left[\sin\theta\right]_0^{\frac{\pi}{2}} = \frac{4}{\pi}$$

In 3 dimensions, a point on the half unit sphere $x^2 + y^2 + z^2 = 1$, $x \geq 0$ has abscissa $\cos\theta$ if it belongs to a circle perpendicular to the x axis of perimeter $2\pi\sin\theta$. Thus we get :

$$3 \cdot \frac{\int_0^{\frac{\pi}{2}} \cos\theta\, 2\pi\, \sin\theta\, d\theta}{\int_0^{\frac{\pi}{2}} 2\pi\, \sin\theta\, d\theta} = 3 \cdot \frac{\int_0^{\frac{\pi}{2}} \sin 2\theta\, d\theta}{2} = \frac{3}{2}$$

We give in Appendix a detailed pseudo code description of the algorithm in two dimensions. The two dimensional orientation tests are replaced by comparison of coordinates denoted by `below`, `above`, `left` or `right` in the pseudo code.

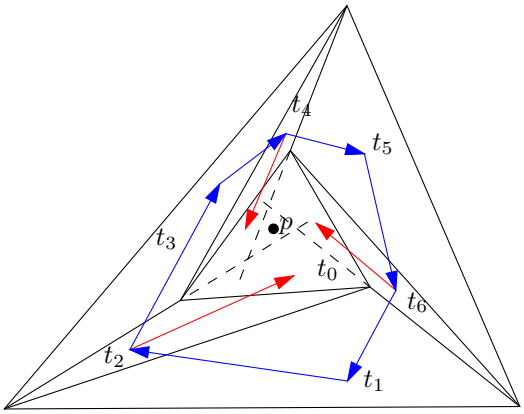

Figure 5: A cycle for the visibility walk.

5 Visibility and stochastic walks

5.1 Description

The *visibility walk* is extremely simple. Let us describe it in 2D. The 3D case is similar, triangles just have to be replaced by tetrahedra and edges by facets. The algorithm starts from a triangle incident to the starting vertex q. Then, for each visited triangle t, the first edge e is tested. If the line supporting e separates t from p, which reduces to a single orientation test, then the next visited triangle is the neighbor of t through e. Otherwise, the second edge is tested in the same way. In case the test for the second edge also fails, then the third edge is tested. The failure of this third test means that the goal has been reached and that t contains p.

In addition to its simplicity, the advantage of this walk is that it does not have to deal with degeneracies. If, for an edge e, p lies on the supporting line of e, then the method will look at the next edge. At least one of the edges of each triangle is such that its supporting line strictly separates the triangle from the query point. The only degeneracies to be considered, namely the different cases when p lies on the boundary of a triangle, occur at the end of the walk, when the goal is reached.

The visibility walk is not completely specified: it depends on the implementation of the triangulation, since there is no intrinsic numbering of the edges of a triangle, no intrinsic definition of the "first" edge. The straight walk can be seen as a possible particular execution of the visibility walk algorithm. This is not the case for the orthogonal walk.

The visibility walk in a Delaunay triangulation always terminates, in any dimension [11]. Unfortunately, for non-Delaunay triangulations, the visibility walk may fall into a cycle, even in 2D, as illustrated by the famous example of Figure 5. Non-Delaunay triangulations (e.g. the constrained Delaunay triangulation) are also interesting in practice and they cannot be eluded. A little bit of randomness can be introduced to avoid infinite loops. As already noticed, the visibility walk depends on the numbering of the

edges of the triangles. Using this degree of freedom, we may choose between different possible visibility walks.

The *stochastic walk* is obtained by replacing the access to the *first* edge of t by the access to a *random* edge of t. For each visited triangle t, a random edge e is tested first. If the line supporting e separates t from p, the next visited triangle is the neighbor of t through e. Otherwise, the other edges of t are tested in some predefined order until an edge separating t from p is found. This ensures that, if the walk enters a cycle of the triangulation, it cannot loop into this cycle forever. The termination of the stochastic walk in any kind of triangulation will be proved in the next section.

The stochastic walk performs 1 to 3 orientation tests in each visited triangle. More precisely, suppose a triangle has only one edge whose supporting line separates it from p, then, this edge is chosen as the first one with probability $1/3$, and only one test is needed, the previous one is chosen with probability $1/3$, and two tests are performed, or the next one is chosen, and three tests are performed. This amounts to $1/3 \cdot 1 + 1/3 \cdot 2 + 1/3 \cdot 3 = 2$. In the case when the triangle has two edges whose supporting lines separate it from p, the number of tests is $2/3 \cdot 1 + 1/3 \cdot 2 = 4/3$. Thus, the average number of orientation tests is less than 2, whereas it is 2 for the straight walk. Similar computations show that in 3D, the average number of tests is less than 2.5, whereas it is 3 for the straight walk.

A variant of the stochastic walk is the *remembering stochastic walk* whose pseudo-code is given in Appendix. In a given triangle, the visibility (stochastic or not) walk can test the edge where it comes from, and thus performs an orientation test that was already performed in the previous visited triangle. This can be avoided by remembering, for each visited triangle, the edge that was just crossed by the walk. Then, before testing an edge, it compares it with the remembered edge. This comparison consists of a constant number of comparison of pointers, as mentioned in Section 2. Computations analogous to the ones done above for the variant without memory lead to an average number of orientation tests less than 1.5 in two dimensions and less than 2 in three dimensions. It is not clear whether remembering the edge and performing the comparisons for each triangle is less expensive in practice than a useless orientation test in some triangles. The two variants will be compared experimentally in Section 6.

5.2 Expected validity of the stochastic walk

Let us analyze the algorithm in dimension d.

Given p, we define the directed graph $\mathcal{G}$, from $\mathcal{T}$, as follows. The nodes of $\mathcal{G}$ are the simplices of $\mathcal{T}$ (we will use the same notation for a simplex and its associated node), and there is an oriented arc from node t to node t' if the corresponding simplices are adjacent through a facet e, in such a way that t' and p lie on the same side of e (see Figure 6).

Lemma 1 *Given two adjacent simplices t and t' such that the arc of $\mathcal{G}$ between node t and node t' is oriented from t to t', the probability that a stochastic path reaching t goes to t' is greater than $\frac{1}{d+1}$.*

Proof: Node t has $d+1$ incident arcs in $\mathcal{G}$. Given an outgoing arc among the arcs incident to t, the probability that this arc is tested first is exactly $\frac{1}{d+1}$. This arc can also

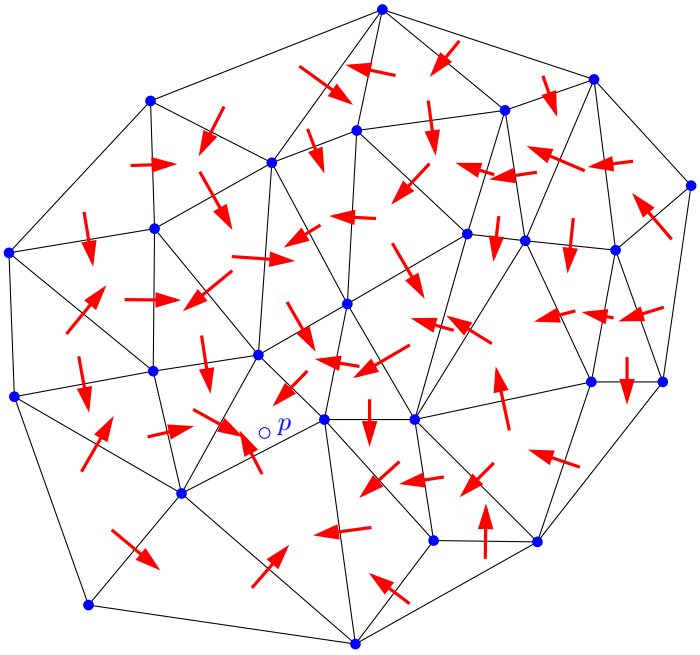

Figure 6: The directed graph $\mathcal{G}$ of neighborhood relationships towards p.

be found after testing several ingoing edges, which only increases the probability of choosing it as the next arc in the path.

Theorem 2 *Given a triangulation $\mathcal{T}$ and a query point p in dimension d, the stochastic walk terminates with probability 1.*

Proof: The out-degree of a node of $\mathcal{G}$ is between 1 and d. As noticed before, the graph $\mathcal{G}$ may have cycles, but we will prove that the stochastic walk cannot cycle forever and will necessarily reach the only sink of the graph $\mathcal{G}$, i.e. the simplex S containing p.

Let us label all the nodes of $\mathcal{G}$ by their distances to S in $\mathcal{G}$, where the distance between a node to S is the minimum number of arcs to be followed to reach S from this node (by the definition of $\mathcal{G}$, there is always a path from any node to S). Then by construction, for any node t of label k, there exists an arc of $\mathcal{G}$ from t to at least one node of label $k-1$. Thus if t is visited, then a node of label $k-1$ is visited with probability higher than $\frac{1}{d+1}$ by Lemma 1.

Assume that a stochastic walk visits N_k nodes of label k. Then $N_{k-1} \geq \frac{N_k}{d+1}$ and by an immediate induction: $N_k \leq (d+1)^k N_0$. This relation clearly proves that $N_0 \neq 0$. So, the walk terminates and reaches S, which is the only node of label 0.

Additionally, this proof yields an exponential bound on the length of the stochastic walk:

$$Cost \leq \sum_{k=0}^{\Delta} N_k \leq \sum_{k=0}^{\Delta} (d+1)^k \leq \frac{(d+1)^{\Delta+1}}{d}$$

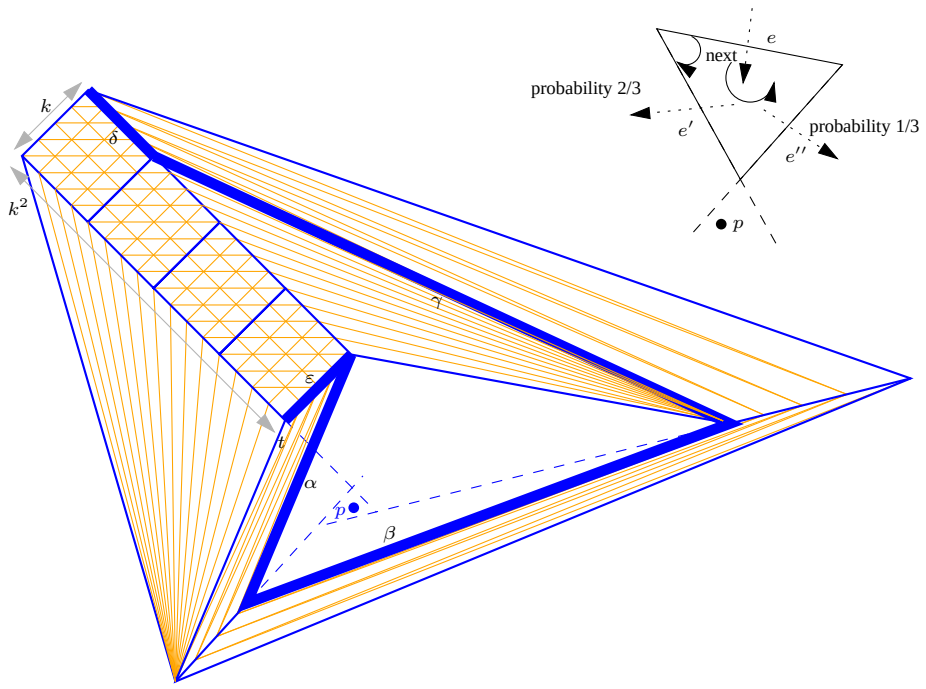

Figure 7: The stochastic walk may have exponential length.

($N_0 = 1$) where Δ is the maximal length (in terms of number of arcs) of a shortest path in $\mathcal{G}$ from any node to S. Since the straight walk is a particular case of visibility walk, Δ is bounded by the longest straight walk in the triangulation, that is $\Delta = O(n)$ in two dimensions and $\Delta = O(n^2)$ in three dimensions.

Unfortunately, for very special configurations of points, this exponential length of the stochastic walk can actually be reached.

The triangulation depicted on Figure 7 consists of one central triangle containing the point p to be located and k layers of cycles around it. These cycles go through a rectangle formed by $k \times k^2$ small squares.

Any triangle having two outgoing arcs in graph $\mathcal{G}$, in this example, is as shown in Figure 7: the ingoing edge e is chosen first with probability $1/3$, then the walk must cross the next edge e', which forces the walk to follow a cycle. e' can also be chosen first with probability $1/3$. So, the walk stays in the cycle with probability $2/3$. The edge e'' allows the walk to leave the cycle. It is crossed only when it is chosen first, which occurs with probability $1/3$.

Let a stochastic walk start in triangle t defined in the figure.
It reaches p through edge α if, for each visited triangle, it chooses the edge out of the cycle, which occurs with probability $(1/3)^k$. It reaches p through edge β if it chooses i times the edge of the cycle, then it chooses the edge in the cycle, then it chooses $k - i$ times the edge out of the cycle. This occurs with probability $k.(2/3).(1/3)^k$.

Analogously, it crosses edge γ with probability $\binom{k+1}{2}.(2/3)^2.(1/3)^k$.
Thus, the walk enters the $k \times k^2$ rectangle by one of the k edges δ with probability $1-(1/3)^k - k.(2/3).(1/3)^k - \binom{k+1}{2}.(2/3)^2.(1/3)^k$.
Let us consider a path entering the rectangle through δ. Giving a tight bound on the probability that such a path goes out of the rectangle through one edge of ε is quite complicated. Let us use a loose bound equal to $(1/2)^{k-1}$. If the path does not go out through ε, then it necessarily reaches its starting triangle t, and using the previous bound, this occurs with probability greater than $1-(1/2)^{k-1}$.
Summarizing, a path starting at t reaches t again with probability greater than

$$\left(1-(1/3)^k - k.(2/3).(1/3)^k - \binom{k+1}{2}.(2/3)^2.(1/3)^k\right)\left(1-(1/2)^{k-1}\right)$$
$$\geq 1 - \frac{2k^2+8k+9}{3^{k+2}} - \frac{1}{2^{k-1}}.$$

So, the expected number of times that t is visited is greater than

$$\sum_{j=0}^{\infty}\left(1 - \frac{2k^2+8k+9}{3^{k+2}} - \frac{1}{2^{k-1}}\right)^j = \frac{1}{\frac{2k^2+8k+9}{3^{k+2}} + \frac{1}{2^{k-1}}}.$$

The shortest cycle from t to t is obtained by traversing the triangles having a vertex on the convex hull of the points. Its length is k^2+2k+6. So, we get an expected length for stochastic walk greater than $\frac{k^2+2k+6}{\frac{2k^2+8k+9}{3^{k+2}}+\frac{1}{2^{k-1}}} \geq 2^{k+1}$.
Moreover, this triangulation has $(k+1)(k^2+1)+2(k+1) \leq (k+1)^3$ vertices. Thus, we proved the following result:

Theorem 3 *There exists a triangulation $\mathcal{T}$ of n points in dimension 2 and a query point p, such that the expected length of the stochastic walk is greater than $2\sqrt[3]{n}$.*

6 Experimental results

6.1 Data sets

All the strategies described above, except the visibility walk, can work on arbitrary triangulations. However, the triangulations that are commonly used in practice are well shaped (most of their triangles satisfy the Delaunay property). The performances on a triangulation having a large number of long edges will be lowered for each strategy but these triangulations are rarely interesting in practice.

So, we experimented different walking strategies for locating points in a Delaunay triangulation of 100.000 or 1.000.000 points evenly distributed in a square or in a cube, as well. For these evenly distributed points, the results are averaged on 5 different sets in each case.

We also experimented on real data, namely a set of 145,300 points in three-dimensional space belonging to the boundary of a 3D object, these points have been measured by a 3D laser scanner on a dental prothesis (courtesy of KREON Industrie).

Delaunay triangulations are a reasonable choice for experimentation, since they are well shaped. However we have made some experiments on a bad shaped triangulation obtained by taking the constrained Delaunay triangulation of 5.000 almost vertical non-intersecting segments obtained by linking 10.000 random points sorted in x coordinates.

6.2 Implementation

The algorithms are coded in C++. The orientation tests use the usual floating point arithmetic and static filtering, which do not have a significant influence on the running time. In order to evaluate the influence of the chosen arithmetic, a few experiments using `LEDA reals` (version 4.2) have also been performed [9].

The walk was performed in the standard way, starting the walk at some known vertex of the triangulation. In the case of Delaunay triangulations, it was also used as a tool in the Delaunay hierarchy [8] which walks in a hierarchy of more and more refined samples; using this method, locating a query involves few ($O(\log n)$) walks visiting a relatively small number of triangles.

To simplify the implementation, degenerate cases in the straight walk are not treated, leading to a non-robust program. However, no problem was encountered during the experiments on the test data sets, which did not contain any set of three collinear (resp four coplanar) points in 2D (resp 3D).

6.3 Results

For each strategy we count the number of visited triangles or tetrahedra ($\sharp\Delta$), the number of full dimensional orientation predicates ($\sharp$orient) and the running time (benchmarks are done on a Pentiun III 935 MHz 512 Mo); times are obtained with the `clock` command, and averaged on 100.000 locations in each triangulation.

Figure 8 shows the results obtained on the Delaunay triangulations, where the visibility walk (without randomness) does not cycle and produces results similar to the stochastic walk, in both basic and remembering versions. On the constrained Delaunay triangulation tested, the visibility walk actually fails in about 0.01 percent of the located points (Figure 9).

On Delaunay triangulations, the running times of all strategies are of the same order. The visiblity and the stochastic walks have a small advantage in terms of running times on the straight and orthogonal walks.

The straight walk has the best performances in terms of visited simplices, both theoretically and experimentally, but it has the worst cost per triangle. Another drawback of the straight walk is that the code is more intricate, especially in three dimensions, and it would be even more intricate if degeneracies were treated.

For walks of large length in terms of visited simplices (i.e. when the hierarchy is not used), the orthogonal walk is fast. In fact it will be the right choice when using expensive arithmetic (e.g. multi-precision exact arithmetic) as confirmed by some experiments using the `real` arithmetic provided by LEDA (see Figure 10).

	Walk			Hierarchy		
	$\sharp\Delta$	$\sharp_{\text{orient}}$	μs	$\sharp\Delta$	$\sharp_{\text{orient}}$	μs
	per point			per point		
100,000 random points						
Stochastic 2D	369	633	154.6	23	48	17.6
Visibility 2D	373	637	145.4	24	46	17.0
Rem. stoch. 2D	369	489	147.1	23	38	17.7
Rem. visib. 2D	373	488	139.7	23	35	17.0
Straight 2D	343	688	162.3	19	42	17.8
Orthogonal 2D	420	3	141.2	26	8	18.9
Stochastic 3D	161	312	98.3	30	70	29.4
Visibility 3D	174	361	101.5	30	74	28.9
Rem. stoch. 3D	161	250	95.9	31	58	29.3
Rem. visib. 3D	174	284	98.7	30	59	28.5
Straight 3D	150	445	109.5	25	75	30.2
Orthogonal 3D	198	12	112.0	42	21	39.7
1,000,000 random points						
Stochastic 2D	1161	1981	613.9	29	61	22.8
Visibility 2D	1174	1998	587.0	30	58	22.0
Rem. stoch. 2D	1161	1534	577.6	28	46	22.5
Rem. visib. 2D	1174	1532	572.2	29	43	22.0
Straight 2D	1082	2165	595.0	25	53	23.0
Orthogonal 2D	1376	3	577.8	34	11	25.0
Stochastic 3D	340	651	240.0	35	80	37.8
Visibility 3D	367	751	258.3	36	85	37.8
Rem. stoch. 3D	340	524	236.0	35	65	37.6
Rem. visib. 3D	367	593	253.6	36	69	37.5
Straight 3D	315	937	263.1	31	90	39.5
Orthogonal 3D	385	16	241.8	48	21	50.1
dental prothesis (145,300 points)						
Stochastic 3D	133	275	75.2	41	94	34.3
Visibility 3D	153	335	80.3	42	100	34.3
Rem. stoch. 3D	133	221	73.0	41	77	33.8
Rem. visib. 3D	153	266	78.5	42	80	34.0
Straight 3D	117	339	75.4	35	103	34.9
Orthogonal 3D	144	39	86.5	74	41	54.6

Figure 8: Running times for the Delaunay triangulation of random and realistic inputs

	Walk		
	$\sharp\Delta$	$\sharp_{\text{orient}}$	μs
	per point		
10.000 points constrained			
Stochastic 2D	1275	1964	191.6
Visibility 2D			loop
Rem. stoch. 2D	1275	1527	188.4
Rem. visib. 2D			loop
Straight 2D	3325	6653	843.3
Orthogonal 2D	3453	3	620.1

Figure 9: Running times for constrained Delaunay triangulation of 5.000 *almost vertical* segments

	Walk	Hierarchy
	μs per point	μs per point
100.000 points		
Rem. stoch. 2D	1170	105
Straight 2D	980	79
Orthogonal 2D	230	49
1.000.000 points		
Rem. stoch. 2D	4300	120
Straight 2D	3500	96
Orthogonal 2D	1000	64

Figure 10: Running times with expensive arithmetic

The experiments show that the stochastic walk is the best choice for implementation, since
— it has good practical performances,
— it does not encounter any problem with degenerate cases and thus it is much simpler to implement
— and it still has a guarantee of validity for non-Delaunay triangulations.

7 Conclusion and open problems

We described four strategies for walking in a triangulation to locate a point: the straight walk, the visibility walk with or without memory, and the orthogonal walk. We studied them from both theoretical and practical points of view.

The best method to implement is the stochastic visibility walk, since it performs experimentally a little bit better than the straight and the orthogonal walks, and since it is easier to code and does not encounter any problem with degenerate cases. The orthogonal walk can also be considered when an expensive arithmetic is used or when a large number of simplices must be traversed.

Open questions remain about the stochastic visibility walk. We showed that it always terminates, but it can have an exponential complexity on cases that are very pathologic, both in the choice of the triangulation and in the choice of the query point. It might be possible to get results under some hypotheses on the triangulation and on the query point: Is the expected complexity in the case of a Delaunay triangulation of n random points in dimension d equal to $\sqrt[d]{n}$? Would it be possible to get an amortized complexity for the successive locations of n points incrementally inserted into a Delaunay triangulation?

References

[1] F. P. Preparata. Planar point location revisited. *Internat. J. Found. Comput. Sci.*, 1(1):71–86, 1990.

[2] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[3] C. L. Lawson. Software for C^1 surface interpolation. In J. R. Rice, editor, *Math. Software III*, pages 161–194. Academic Press, New York, NY, 1977.

[4] P. J. Green and R. R. Sibson. Computing Dirichlet tessellations in the plane. *Comput. J.*, 21:168–173, 1978.

[5] A. Bowyer. Computing Dirichlet tesselations. *Comput. J.*, 24:162–166, 1981.

[6] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 274–283, 1996.

[7] C. Lemaire. *Triangulation de Delaunay et arbres multidimensionnels*. Thèse de doctorat en sciences, École des Mines de St-Etienne, France, 1997.

[8] Olivier Devillers. The Delaunay hierarchy. This volume.

[9] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[10] H. Edelsbrunner. An acyclicity theorem for cell complexes in d dimensions. *Combinatorica*, 10(3):251–260, 1990.

[11] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. On sorting triangles in a Delaunay tessellation. *Algorithmica*, 6:522–532, 1991.

[12] Paul-Louis George and Houman Borouchaki. *Triangulation de Delaunay et maillage. Applications aux éléments finis*. Hermes, Paris, France, 1997.

[13] P. Bose and L. Devroye. Intersections with random geometric objects. Technical report, School of Computer Science, McGill University, 1995. Manuscript.

[14] Luc Devroye, Ernst Peter Mücke, and Binhai Zhu. A note on point location in Delaunay triangulations of random points. *Algorithmica*, 22:477–482, 1998.

[15] Jean-Daniel Boissonnat, Olivier Devillers, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 11–18, 2000.

Algorithm *2D Straight Walk(q,p)*

```
// traverses the triangulation T, following the line segment
from q to p.
// t = qrl is a triangle of T.
if orientation(rqp)<0 while orientation(lqp)<0 {
     r=l;
     t=neighbor(t through ql);
     l=vertex of t, l≠q, l≠r; }
else do {
     l=r;
     t=neighbor(t through qr);
     r=vertex of t, r≠q, r≠l;
     } while orientation(rqp)<0;
// end of initialization step
// now qp has r on its right and l on its left.
while orientation(prl)<0 {
  t = neighbor(t through rl);
  s = vertex of t, s≠r, s≠l;
  if orientation(sqp)<0 r=s; else l=s; }
// t contains p.
```

Algorithm *3D Straight Walk(q,p)*

```
// traverses the triangulation 𝒯,
// following the line segment from q to p.
// t = uvwq is a tetrahedron of 𝒯.
if orientation(vuqp)>0 while orientation(wuqp)>0 {
     v=w;
     t=neighbor(t through quw);
     w=vertex of t, w≠u, w≠v, w≠q; }
else do {
     w=v;
     t=neighbor(t through quv);
     v=vertex of t, v≠u, v≠w, v≠q;
     } while orientation(vuqp)<0;
// now v and w lie on opposite sides of plane uqp,
// vuqp is positively oriented and wuqp negatively.
while orientation(vwqp)>0 {
  t = neighbor(t through qvw);
  s = vertex of t, s≠v, s≠w, s≠q;
  if orientation(suqp)>0 v=s; else w=s; }
u = vertex of t, s≠v, s≠w, s≠q;
// end of initialization step,
// qp intersects triangle uvw,
// wvqp, vuqp and uwqp are positively oriented.
while orientation(uwvp)>0 {
  t = neighbor(t through uvw);
  s = vertex of t, s≠u, s≠v, s≠w;
  if orientation(usqp)>0 // qp does not intersect triangle
usw,
     if orientation(vsqp)>0 // qp intersects triangle vsw,
       u=s;
     else // qp intersects triangle usv,
       w=s;
  else // qp does not intersect triangle usv,
     if orientation(wsqp)>0 // qp intersects triangle usw,
       v=s;
     else // qp intersects triangle vsw,
       u=s;
// t contains p.
```

Algorithm *2D Orthogonal Walk(q,p)*

```
// traverses the triangulation 𝒯, using the orthogonal walk
from q to p,
// t = qrl is a triangle of 𝒯.  wlog, we assume p is above and
to the right of q.
α=point(xp,yq);
if r below q while l below q {
     r=l; t=neigbor(t through ql); l=vertex of t≠qr;}
else do {
     l=r; t=neigbor(t through qr); r=vertex of t≠ql;
  } while r above q;
// q has r below and l above.
while ((r and l at left of α) or orientation(αrl)<0){
     t = neighbor(t through rl);
     s = vertex of t≠rl;
     if s above q l=s; else r=s; }
// α inside t
l = vertex of t≠rl;
r = vertex of t≠rl;
// p has r at right and l at left.
while ((r and l below p) or orientation(prl)<0) {
     t = neighbor(t through rl);
     s = vertex of t≠rl;
     if s at left of p l=s; else r=s;}
// t contains p.
```

Algorithm *Remembering Stochastic Walk(q,p)*

```
// traverses the triangulation 𝒯, using the remembering stochastic
walk
// from q to p.  t = qrl is a triangle of 𝒯.
previous=t; end=false;
while (not end) {
  e = random edge of t;
  if (p not neighbor of previous through e)
    and (p on the other side of e)
         {previous=t;t=neighbor(t through e);}
  else {
    e = next edge of t;
    if (p not neighbor of previous through e)
      and (p on the other side of e)
        {previous=t;t=neighbor(t through e);}
    else {
      e = next edge of t;
      if (p not neighbor of previous through e)
        and (p on the other side of e)
        {previous=t;t=neighbor(t through e);}
      else end=true;
    }
  }
}
// t contains p.
```